Controladores de Ratón

Alan Cox

alan@redhat.com

Controladores de Ratón

by Alan Cox

Table of Contents

List of Tables

Chapter 1. Introducción

> **Publicación Anterior:** Algunas partes de este documento aparecieron primero en Linux Magazine bajo una exclusividad de noventa dias.

Los ratones son conceptualmente uno de los interfaces de dispositivos más simples en el sistema operativo Linux. No todos los ratones son manejados por el núcleo. Es vez de eso, hay una abstracción de dos capas.

Los controladores de ratón del núcleo y los controladores del espacio de usuario para los ratones serie son todos administrados por un demonio del sistema llamado gpm - el controlador de propósito general de ratón. gpm maneja la acción de cortar y pegar en los textos de las consolas. Suministra una biblioteca general para aplicaciones que conocen el ratón y administra la compartición de los servicios del ratón con la interfaz de usuario del X Window System.

Algunas veces un ratón habla un protocolo suficientemente complicado como para que sea manejado por el propio Gpm. La mayoría de los controladores de ratón siguen una interfaz común llamada protocolo de bus del ratón.

Cada lectura de un dispositivo de una interfaz del bus de ratón retorna un bloque de datos. Los tres primeros bytes de cada lectura están definidos de la siguiente forma:

Table 1-1. Codificación de Datos del Ratón

Byte 0	0x80 + los botones actualmente pulsados.
Byte 1	Un valor con signo para el desplazamiento en la posición X
Byte 2	Un valor con signo para el desplazamiento en la posición Y

Una aplicación puede escoger leer más de 3 bytes. El resto de los bytes serán cero, o quizás opcionalmente retornen alguna información específica del dispositivo.

Los valores de la posición son truncados si es que exceden del rango de los 8 bits (que es -127 <= delta <= 127). Como el valor -128 no encaja en un byte no es permitido.

Los botones son numerados de izquierda a derecha como 0, 1, 2, 3.. y cada botón establece el bit relevante. Por lo tanto un usuario presionando los botonoes de la izquierda y de la derecha en un ratón de tres botones establecerán los bits 0 y 2.

Todos los ratones están requeridos a soportar la operación `poll`. Sería algo verdaderamente muy bonito si todos los usuarios de un controlador de un dispositivo usaran `poll` para esperar a que tuvieran lugar

los eventos.

Finalmente el soporte asíncrono de E/S de los ratonoes. Este es un tópico que todavía no hemos cubierto pero que explicaré más tarde, después de mirar en un controlador simple de ratón.

Chapter 2. Un controlador simple de ratón

Primero necesitaremos inicializar las funciones para nuestro dispositivo ratón. Para mantener esto simple, nuestro dispositivo imaginario de ratón tiene tres puertos de E/S en las direcciones de E/S 0x300 y siempre vivirá en la interrupción 5. Los puertos serán la posición X, la posición Y y los botones, en este orden.

```
#define OURMOUSE_BASE        0x300

static struct miscdevice our_mouse = {
        OURMOUSE_MINOR, "ourmouse", &our_mouse_fops
};

__init ourmouse_init(void)
{

        if(check_region(OURMOUSE_BASE, 3))
                return -ENODEV;
        request_region(OURMOUSE_BASE, 3, "ourmouse");

        misc_register(&our_mouse);
        return 0;
}
```

El miscdevice es nuevo aquí. Linux normalmente divide los dispositivos por su número mayor, y cada dispositivo tiene 256 unidades. Para cosas como los ratones esto es extremadamente derrochador para la existencia de un dispositivo que es usado para acumular todos los dispositivos individuales sueltos que las computadoras tienden a tener.

Los números menores en este espacio son asignados por un código central, aunque puedes mirar en el el archivo `Documentation/devices.txt` del núcleo y coger uno libre para un uso de desarrollo. Este archivo de núcleo también contiene instrucciones para registrar un dispositivo. Esto puede cambiar con respecto al tiempo y es, por lo tanto, una buena idea obtener primero una copia actualizada de este archivo.

Nuestro código es entonces bastante simple. Chequeamos si nadie más ha tomado nuestro espacio de direcciones. Habiéndolo hecho, lo reservamos para asegurarnos de que nadie pisa a nuestro dispositivo mientras estamos probando otros dispositivos del bus ISA. Ya que una prueba quizás confunda a nuestro dispositivo.

Entonces le decimos al controlador misc que queremos nuestro propio número menor. También cogemos nuestro nombre (que es usado en `/proc/misc`) y establecemos las operaciones de archivo que van a ser usadas. Las operaciones de archivo trabajan exactamente como las operaciones de archivo para registrar un dispositivo de carácter normal. El dispositivo misc simplemente actúa como redirector para las peticiones.

Lo siguiente, en orden a ser capaz de usar y probar nuestro propio código, es que necesitamos añadir algún código de módulo para soportarlo. Esto también es bastante simple:

```
#ifdef MODULE

int init_module(void)
{
        if(ourmouse_init()<0)
                return -ENODEV:
        return 0;
}

void cleanup_module(void)
{
        misc_deregister(&our_mouse);
        free_region(OURMOUSE_BASE, 3);
}


#endif
```

El código del módulo suministra las dos funciones normales. La función `init_module` es llamada cuando el módulo es cargado. En nuestro caso simplemente llama a la función de inicialización que escribimos y retorna un error si esta falla. Esto asegura que el módulo sólo será cargado si fue correctamente configurado.

La función `cleanup_module` es llamada cuando el módulo es descargado. Devolvemos nuestra entrada de dispositivo misceláneo, y entonces liberamos nuestros recursos de E/S. Si no liberamos nuestros recursos de E/S entonces la siguiente vez que el módulo es cargado pensaremos que alguien tiene este espacio de E/S.

Una vez que `misc_deregister` ha sido llamada cualquier intento de abrir el dispositivo del ratón fallará con el error ENODEV (No tal dispositivo).

Lo siguiente que necesitamos es rellenar nuestras operaciones de archivo. Un ratón no necesita muchas de estas. Necesitamos suministrar open (abrir), release (liberar), read (leer) y poll (encuesta). Esto hace una bonita y simple estructura:

```
struct file_operations our_mouse_fops = {
        owner: THIS_MODULE,           /* Automática administración de uso */
        read:  read_mouse,            /* Puedes leer un ratón */
        write: write_mouse,           /* Esto debería de hacer un montón */
        poll:  poll_mouse,            /* Encuesta */
        open:  open_mouse,            /* Llamado en open */
        release: close_mouse,         /* Llamado en close */
};
```

No hay nada particularmente especial necesitado aquí. Suministramos funciones para todas las operaciones relevantes o requiridas y algunas pocas más. No hay nada que nos pare para suministrar una función ioctl para este ratón. Verdaderamente si tienes un ratón configurable quizás sea muy apropiado suministrar interfaces de configuración a través de llamadas ioctl.

La sintaxis que usamos no es la del C estándar. GCC suministra la habilidad de inicializar campos por el nombre, y esto generalmente hace la tabla de métodos mucho más fácil de leer y contar a través de los punteros NULL y de recordar el orden a mano.

El dueño del campo es usado para administrar el bloqueo de la carga y descarga de un módulo. Esto es obviamente importante para que un módulo no sea descargado mientras esté siendo usado. Cuando tu dispositivo es abierto, el módulo especificado por "owner" es bloqueado. Cuando el módulo es finalmente liberado es desbloqueado.

Las rutinas open y close necesitan administrar el habilitamiento y deshabilitamiento de las interrupciones para el ratón y también el parar el ratón siendo descargado cuando no se requiere más.

```
static int mouse_users = 0;               /* Cuenta de Usuarios */
static int mouse_dx = 0;                  /* Cambios de Posición */
static int mouse_dy = 0;
static int mouse_event = 0;               /* El ratón se movió */

static int open_mouse(struct inode *inode, struct file *file)
{
        if(mouse_users++)
                return 0;

        if(request_irq(mouse_intr, OURMOUSE_IRQ, 0, "ourmouse", NULL))
        {
                mouse_users--;
                return -EBUSY;
        }
        mouse_dx = 0;
        mouse_dy = 0;
        mouse_event = 0;
        mouse_buttons = 0;
return 0;
}
```

La función open tiene que hacer una pequeña cantidad de tareas domésticas. Mantenemos una cuenta del número de veces que el ratón está abierto. Esto es porque no queremos pedir la interrupción múltiples veces. Si el ratón tiene por lo menos un usuario, es configurado y simplemente añadimos el usuario a la cuenta y retornamos 0 para indicar que tuvo éxito.

Cogemos la interrupción y entonces comienzan las interrupciones del ratón. Si la interrupción ha sido apropiada por otro controlador entonces `request_irq` fallará y retornará un error. Si fuimos capaces de

compartir una línea de interrupción deberíamos de especificar `SA_SHIRQ` en vez de `zero`. Siempre que todo el mundo que coga una interrupción establezca este flag, compartirán la línea. PCI puede compartir interrupciones, ISA normalmente no.

Hacemos las tareas domésticas. Hacemos a la actual posición del ratón el punto de comienzo para los cambios acumulados y declaramos que no ha pasado nada desde que el controlador del ratón ha sido abierto.

La función release (liberar) necesita desenrollar todas estas:

```
static int close_mouse(struct inode *inode, struct file *file)
{
        if(--mouse_users)
                return 0;
        free_irq(OURMOUSE_IRQ, NULL);
        return 0;
}
```

Descontamos un usuario y siempre que todavía halla otros usuarios que no necesiten acciones adicionales. La última persona cerrando el ratón causa que liberemos la interrupción. Esto para las interrupciones desde el ratón usando nuestro tiempo de CPU, y asegura que el ratón puede ser ahora descargado.

Podemos rellenar el manejador de escritura en este punto como la función write para la que nuestro ratón simplemente declina permitir escrituras:

```
static ssize_t write_mouse(struct file *file, const char *buffer, size_t
                                count, loff_t *ppos)
{
        return -EINVAL;
}
```

Esto es bastante auto-explicativo. Siempre que escribes dirán que era una función inválida.

Para hacer que las funciones read y poll trabajen tenemos que considerar como manejar las interrupciones de ratón.

```
static struct wait_queue *mouse_wait;
static spinlock_t mouse_lock = SPIN_LOCK_UNLOCKED;

static void ourmouse_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        char delta_x;
        char delta_y;
        unsigned char new_buttons;
```

```
        delta_x = inb(OURMOUSE_BASE);
        delta_y = inb(OURMOUSE_BASE+1);
        new_buttons = inb(OURMOUSE_BASE+2);

        if(delta_x || delta_y || new_buttons != mouse_buttons)
        {
                /* Algo ha pasado */

                spin_lock(&mouse_lock);
                mouse_event = 1;
                mouse_dx += delta_x;
                mouse_dy += delta_y;
                mouse_buttons = new_buttons;
                spin_unlock(&mouse_lock);

                wake_up_interruptible(&mouse_wait);
        }
}
```

El manejador de interrupciones lee el status del ratón. La siguiente cosa que hacemos es chequear cuando algo ha cambiado. Si el ratón estaba listo sólo nos debería de interrumpir si algo a cambiado, pero asumamos que nuestro ratón es estúpido, tal como tienden a ser la mayoria de los ratones.

Si el ratón ha cambiado necesitamos actualizar las variables de estado. Lo que no queremos es que las funciones del ratón leyendo estas variables lean durante un cambio. Añadimos un spinlock que protega estas variables mientras jugamos con ellas.

Si ha ocurrido un cambio también necesitamos despertar a los procesos que estén durmiendo, por lo tanto añadimos una llamada wakeup (despertar) y una wait_queue para usar cuando queremos esperar un evento de ratón.

Ahora que tenemos la cola de espera podemos implementar la función poll para el ratón de una forma relativamente fácil:

```
static unsigned int mouse_poll(struct file *file, poll_table *wait)
{
        poll_wait(file, &mouse_wait, wait);
        if(mouse_event)
                return POLLIN | POLLRDNORM;
        return 0;
}
```

Esto es un código de encuesta bastante estándar. Primero añadimos la cola de espera a la lista de colas que queremos monitorizar para un evento. Lo segundo es chequear si ha ocurrido un evento. Nosotros sólo tenemos un tipo de evento - el flag `mouse_event` nos dice que algo ha pasado. Conocemos que esto

sólo pueden ser datos del ratón. Retornamos las flags indicando entrada y realizaremos una lectura normal.

Quizás te asombres de lo que pasa si la función retorna diciendo 'todavía no ocurrió un evento'. En esto caso el despertar de la cola de espera que añadimos a la tabla poll caurará que la función sea llamada otra vez. Eventualmente despertaremos y tendremos un evento listo. En este punto la llamada `poll` puede regresar al usuario.

Después de que poll finalice, el usuario querrá leer los datos. Ahora necesitamos pensar cómo trabajará nuestra función `mouse_read`:

```
static ssize_t mouse_read(struct file *file, char *buffer,
                size_t count, loff_t *pos)
{
        int dx, dy;
        unsigned char button;
        unsigned long flags;
        int n;

        if(count<3)
                return -EINVAL;

        /*
          *        Espera por un evento
         */

        while(!mouse_event)
        {
                if(file->f_flags&O_NDELAY)
                        return -EAGAIN;
                interruptible_sleep_on(&mouse_wait);
                if(signal_pending(current))
                        return -ERESTARTSYS;
        }
```

Empezamos validando que el usuario está leyendo suficientes datos. Podríamos manejar lecturas parciales si quisiéramos, pero esto no es terriblemente útil y los controladores de los ratones no se preocupan de intentarlo.

Acto seguido esperamos que ocurra un evento. El bucle es bastante estándar en Linux para la espera de un evento. Habiendo chequeado que el evento todavía no ha ocurrido, entonces chequeamos si un evento está pendiente y si no es así necesitamos dormir.

Un proceso de usuario puede establecer la flag `O_NDELAY` en un archivo para indicar que desea comunicar inmediatamente si no hay algún evento pendiente. Chequeamos esto y le damos el error apropiado si es así.

A continuación dormimos hasta que el ratón o una señal nos despierte. Una señal nos despertará si hemos usado `wakeup_interruptible`. Esto es importante, ya que significa que un usuario puede matar procesos que estén esperando por el ratón - propiedad limpia y deseable. Si somos interrumpidos salimos de la llamada y el núcleo, entonces, procesará las señales y quizás reinicialice la llamada otra vez - desde el principio.

Este código contiene un fallo clásico de Linux. Todo será revelado después en este articulo, al igual que las explicaciones de cómo eliminarlas.

```
        /* Coge el evento */

        spinlock_irqsave(&mouse_lock, flags);

        dx = mouse_dx;
        dy = mouse_dy;
        button = mouse_buttons;

        if(dx<=-127)
                dx=-127;
        if(dx>=127)
                dx=127;
        if(dy<=-127)
                dy=-127;
        if(dy>=127)
                dy=127;

        mouse_dx -= dx;
        mouse_dy -= dy;

        if(mouse_dx == 0 && mouse_dy == 0)
                mouse_event = 0;

        spin_unlock_irqrestore(&mouse_lock, flags);
```

Esta es la siguiente etapa. Habiendo establecido que hay un evento viniendo, lo capturamos. Para asegurarnos de que el evento no está siedo actualizado cuando lo capturamos también tomamos el spinlock y esto previene las actualizaciones paralelas. Destacar que aquí usamos `spinlock_irqsave`. Necesitamos deshabilitar las interrupciones en el procesador local o en otro caso sucederán cosas malas.

Lo que ocurrirá es que cogeremos el spinlock. Mientras tenemos el bloqueo ocurrirá una interrupción. En este pundo nuestro manejador de interrupciones intentará coger el spinlock. El se sentará en un bucle esperando por la rutina de lectura para que libere el bloqueo. De cualquier forma como estamos sentados en un bucle en el manejador de interrupciones nunca liberaremos el bloqueo. La máquina se cuelga y el usuario se trastorna.

Bloqueando la interrupción en este procesador nos aseguramos de que el mantener el bloqueo siempre nos devolverá el bloqueo sin hacer un deadlocking.

También hay un pequeño truco en el mecanismo de reporte. Sólo podemos reportar un movimiento de 127 por lectura. En todo caso no queremos perder información lanzando movimientos adicionales. En vez de esto, nos mantenemos retornando tanta información como sea posible. Cada vez que retornamos un reporte quitamos la cantidad de movimiento pendiente en `mouse_dx` y `mouse_dy`. Eventualmente cuando estas cuentas llegan a cero, limpiamos el flag `mouse_event` porque ya no queda nada que reportar.

```
        if(put_user(button|0x80, buffer))
                return -EFAULT;
        if(put_user((char)dx, buffer+1))
                return -EFAULT;
        if(put_user((char)dy, buffer+2))
                return -EFAULT;

        for(n=3; n < count; n++)
                if(put_user(0x00, buffer+n))
                        return -EFAULT;

        return count;
}
```

Finalmente tenemos que poner los resultados en el buffer suministrado por el usuario. No podemos hacer esto mientras mantenemos el bloqueo, ya que una escritura a la memoria de usuario quizás duerma. Por ejemplo, la memoria de usuario quizás esté residiendo en disco en este instante. Entonces hicimos nuestra computación de antemano y ahora copiamos los datos. Cada `put_user call` está rellenando en una byte del buffer. Si retorna un error nosotros informamos al programa que nos está pasando un buffer inválido y abortamos.

Habiendo escrito los datos vaciamos el resto del buffer que leimos y reportamos que la lectura tuvo éxito.

Chapter 3. Depurando el Controlador del Ratón

Ahora tenemos un controlador de ratón usable bastante perfecto. Si realmente fueras a probarlo y usarlo en todos los casos eventualmente encontrarías un par de problemas con el. Unos pocos programas no trabajarán con ya que todavía no soporta E/S asíncrona.

Primero déjanos mirar los fallos. El más obvio no es realmente un fallo del controlador sino un fallo al considerar las consecuencias. Imagínate que accidentalmente golpees fuerte el ratón y lo envíes deslizándose sobre la mesa. La rutina de interrupción del ratón añadirá todo el movimiento y lo reportará en pasos de 127 hasta que lo haya reportado todo. Claramente hay un punto lejano desde el cual el valor del movimiento del ratón no es reportado. Necesitamos añadir esto como un límite al manejador de interrupciones:

```
static void ourmouse_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        char delta_x;
        char delta_y;
        unsigned char new_buttons;

        delta_x = inb(OURMOUSE_BASE);
        delta_y = inb(OURMOUSE_BASE+1);
        new_buttons = inb(OURMOUSE_BASE+2);

        if(delta_x || delta_y || new_buttons != mouse_buttons)
        {
                /* Algo ha pasado */

                spin_lock(&mouse_lock);
                mouse_event = 1;
                mouse_dx += delta_x;
                mouse_dy += delta_y;

                if(mouse_dx < -4096)
                        mouse_dx = -4096;
                if(mouse_dx > 4096)
                        mouse_dx = 4096;

                if(mouse_dy < -4096)
                        mouse_dy = -4096;
                if(mouse_dy > 4096)
                        mouse_dy = 4096;

                mouse_buttons = new_buttons;
                spin_unlock(&mouse_lock);

                wake_up_interruptible(&mouse_wait);
        }
}
```

Añadiendo estos chequeos limitamos el rango de movimiento acumulado a algo sensible.

El segundo fallo es un poco más disimulado, y quizás porque es un fallo común. Recuerda, dije que esperando en el bucle por el manejador de lecturas tenía un fallo. Piensa en qué pasa cuando ejecutamos:

```
while(!mouse_event)
{
```

y una interrupción ocurre aquí, en este punto. Esto causa un movimento del ratón y despierta la cola.

```
        interruptible_sleep_on(&mouse_wait);
```

Ahora dormimos en la cola. Perdimos el despertar y la aplicación no verá el evento hasta que ocurra el siguiente evento del ratón. Esto llevará justamente a la instancia suelta cuando un botón del ratón se retrasa. Las consecuencias para el usuario serán bastante indetectables con un controlador de ratón. Con otros controladores este fallo podría ser mucho más severo.

Hay dos formas de solucionar esto. La primera es deshabilitar las interrupciones mientras el testeo y mientras que dormimos. Esto funciona porque cuando una tarea duerme cesa de deshabilitar las interrupciones, y cuando se reinicia las deshabilita otra vez. Nuestro código entonces se convierte en:

```
save_flags(flags);
cli();

while(!mouse_event)
{
        if(file->f_flags&O_NDELAY)
        {
                restore_flags(flags);
                return -EAGAIN;
        }
        interruptible_sleep_on(&mouse_wait);
        if(signal_pending(current))
        {
                restore_flags(flags);
                return -ERESTARTSYS;
        }
}
restore_flags(flags);
```

Esta es la aproximación bruta. Funciona pero significa que gastamos un montón de tiempo adicional cambiando las interrupciones de habilitadas a deshabilitadas. También afecta a las interrupciones globalmente y tiene malas propiedades en máquinas multiprocesadores donde el apagar las interrupciones no es una operación simple, sino que significa hacerlo en cada procesador, esperando por ellos para que deshabiliten las interrupciones y repliquen.

El problema real es la carrera entre la prueba de eventos y el dormir. Podemos eliminar esto usando directamente las funciones de planificación. Realmente esta es la forma que generalmente deberíamos de usar para una interrupción.

```
struct wait_queue wait = { current, NULL };

add_wait_queue(&mouse_wait, &wait);
set_current_state(TASK_INTERRUPTIBLE);

while(!mouse_event)
{
        if(file->f_flags&O_NDELAY)
        {
                remove_wait_queue(&mouse_wait, &wait);
                set_current_state(TASK_RUNNING);
                return -EWOULDBLOCK;
        }
        if(signal_pending(current))
        {
                remove_wait_queue(&mouse_wait, &wait);
                current->state = TASK_RUNNING;
                return -ERESTARTSYS;
        }
        schedule();
        set_current_state(TASK_INTERRUPTIBLE);
}

remove_wait_wait(&mouse_wait, &wait);
set_current_state(TASK_RUNNING);
```

A primera vista esto probablemente parezca magia profunda. Para entender cómo trabaja esto necesitas entender cómo trabajan la planificación y los eventos en Linux. Teniendo un buen dominio de esto es una de las claves para escribir controladores de dispositivos eficientes y claros.

`add_wait_queue` hace lo que su nombre sugiere. Añade una entrada a la lista `mouse_wait`. La entrada en este caso es la entrada para nuestro proceso actual (`current` es el puntero de la tarea actual).

Por lo tanto, empezamos añadiendo una entrada para nosotros mismos en la lista `mouse_wait`. Esto de cualquier forma no nos pone a dormir. Meramente estamos unidos a la lista.

A continuación establecemos nuestro status a `TASK_INTERRUPTIBLE`. Otra vez esto no significa que no estamos dormidos. Este flag dice lo que debería de pasar la siguiente vez que el proceso duerma. `TASK_INTERRUPTIBLE` dice que el proceso no debería de ser replanificado. Él se ejecutará desde ahora hasta que duerma y entonces necesitará ser despertado.

La llamada `wakeup_interruptible` en el manejador de interrupciones puede ahora ser explicada con más detalle. Esta función es también muy simple. Va a través de la lista de procesos en la tarea que le es

dada y cualquiera que esté marcada como `TASK_INTERRUPTIBLE` la cambia a `TASK_RUNNING` y dice al núcleo que son ejecutables nuevos procesos.

Detrás de todos los envoltorios en el código original lo que está sucediendo es esto:

1. Nos añadimos nosotros mismos a la cola de espera del ratón
2. Nos marcamos como durmiendo
3. Preguntamos al núcleo para planificar tareas otra vez
4. El núcleo ve que estamos durmiento y planifica algún otro.
5. La interrupción del ratón establece nuestro estado a `TASK_RUNNING` y destaca que el núcleo debería replanificar tareas
6. El núcleo ve que estamos ejecutándonos otra vez y continúa nuestra ejecución

Esto es porque funciona la aparentemente magia. Porque nos marcamos como `TASK_INTERRUPTIBLE` y nos añadimos a la cola antes de chequear si hay eventos pendientes, la condición de carrera es eliminada.

Ahora si ocurre una interrupción después de que chequeemos el estado de la cola y antes de llamar a la función `schedule` en orden a dormir, las cosas resultan. En vez de perder un evento, estamos volviendo a establecer `TASK_RUNNING` por la interrupción del ratón. Todavía llamamos a `schedule` pero el continuará ejecutando nuestra tarea. Volvemos a través del bucle y esta vez quizás exista un evento.

No habrá siempre un evento. Entonces nos volveremos a establecer a `TASK_INTERRUPTIBLE` antes de continuar el bucle. Otro proceso haciendo una lectura quizás haya limpiado el flag de eventos y si es así necesitaremos regresar a dormir otra vez. Eventualmente obtendremos nuestro evento y salimos.

Finalmente cuando salimos del bucle nos quitamos de la cola `mouse_wait`, ya que no estamos más interesados en eventos del ratón, y ahora nos volvemos a establecer a `TASK_RUNNABLE` ya que todavía no queremos ir a dormir otra vez.

> **Nota:** Este no es un tópico fácil. No tengas miedo de releer la descripción unas pocas veces y también de mirar en otros controladores de dispositivos para ver si funciona. Finalmente si todavía no puedes cogerlo, puedes usar el código como modelo para escribir otros controladores de dispositivos y confiar en mí.

Chapter 4. E/S Asíncrona

Esto deja la característica perdida - E/S Asíncrona. Normalmente los programas UNIX usan la llamada `poll` (o su forma variante `select`) para esperar a que ocurra un evento en uno de los múltiples dispositivos de entrada o salida. Este modelo trabaja bien para la mayoría de las tareas porque las esperas `poll` y `select` para un evento no son convenientes para tareas que están continuamente haciendo trabajo computacional. Tales programas realmente quieren que el núcleo les golpee cuando pasa algo en vez de mirar por los eventos.

Poll es semejante a tener una fila de luces delante de tí. Puedes ver en un instante cuales de ellas están encendidas. No puedes, de cualquier forma, hacer nada útil mientras las estás mirando. La E/S asíncrona usa señales que trabajan más bien como un timbre. Es vez de mirar, dice que algo se ha manifestado.

La E/S asíncrona envía la señal SIGIO al proceso de usuario cuando ocurre el evento de E/S. En este caso esto significa cuando la gente mueve el ratón. La señal SIGIO causa que el proceso de usuario salga a su manejador de señales y ejecute el código en ese manejador antes de regresar a lo que estuviera haciendo previamente. Esta es la aplicación equivalente a un manejador de interrupciones.

La mayor parte del código necesitado para esta operación es común a todos los usuarios. El núcleo suministra un conjunto simple de funciones para administrar la E/S asíncrona.

Nuestro primer trabajo es permitir a los usuarioes establecer E/S asíncrona en el manejadores de archivos. Para hacer esto necesitamos añadir una nueva funciónn a la tabla de operaciones de archivo para nuestro ratón:

```
struct file_operations our_mouse_fops = {
        owner: THIS_MODULE
        read:  read_mouse,      /* Puedes leer un ratón */
        write: write_mouse,     /* Esto no hará mucho */
        poll:  poll_mouse,      /* Encuesta */
        open:  open_mouse,      /* Llamado en open */
        release: close_mouse,   /* Llamado en close */
        fasync: fasync_mouse,   /* E/S asíncrona */
};
```

Una vez que hemos instalado esta entrada, el núcleo conoce que soportamos E/S asíncrona y permitirá todas las operaciones relevantes en el dispositivo. Siempre que un usuario añade o quita la notificación de E/S asíncrona de un manejador de archivos, llama a nuestra rutina `fasync_mouse` que acabamos de añadir. Esta rutina usa las funciones de ayuda para mantener actualizada la cola de manejadores:

```
static struct fasync_struct *mouse_fasync = NULL;

static int fasync_mouse(int fd, struct file *filp, int on)
{
         int retval = fasync_helper(fd, filp, on, &mouse_fasync);
```

```
         if (retval < 0)
                 return retval;
        return 0;
}
```

La fasync helper añade y borra entradas administrando la lista suministrada. También necesitamos quitar entradas de esta lista cuando es cerradi el archivo. Esto requiere añadir una línea a nuestra función close:

```
static int close_mouse(struct inode *inode, struct file *file)
{
        fasync_mouse(-1, file, 0)
        if(--mouse_users)
                return 0;
        free_irq(OURMOUSE_IRQ, NULL);
        MOD_DEC_USE_COUNT;
        return 0;
}
```

Cuando cerramos el archivo podemos llamar a nuestro propio manejador fasync como si el usuario pidiera que este archivo cesara de ser usado para E/S asíncrona. Esto aproximadamente limpia cualesquiera finales perdidos. Seguramente no esperamos por la llegada de una señal para un archivo que no existirá más.

En este punto, el controlador del ratón soporta todas las operaciones de E/S asíncrona, y las aplicaciones usándolas no fallarán. Estas de todas formas no trabajarán todavía. Necesitamos realmente enviar las señales. Otra vez el núcleo suministra una función para manejar esto.

Actualizamos un poco nuestro manejador de interrupciones:

```
static void ourmouse_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        char delta_x;
        char delta_y;
        unsigned char new_buttons;

        delta_x = inb(OURMOUSE_BASE);
        delta_y = inb(OURMOUSE_BASE+1);
        new_buttons = inb(OURMOUSE_BASE+2);

        if(delta_x || delta_y || new_buttons != mouse_buttons)
        {
                /* Algo ha pasado */

                spin_lock(&mouse_lock);
                mouse_event = 1;
                mouse_dx += delta_x;
```

```
                mouse_dy += delta_y;

                if(mouse_dx < -4096)
                        mouse_dx = -4096;
                if(mouse_dx > 4096)
                        mouse_dx = 4096;

                if(mouse_dy < -4096)
                        mouse_dy = -4096;
                if(mouse_dy > 4096)
                        mouse_dy = 4096;

                mouse_buttons = new_buttons;
                spin_unlock(&mouse_lock);

                /* Ahora hacemos E/S asíncrona */
                kill_fasync(&mouse_fasync, SIGIO);

                wake_up_interruptible(&mouse_wait);
        }
}
```

El nuevo código simplemente llama a la rutina `kill_fasync` suminstrada por el núcleo si la cola no está vacía. Esto envía la señal requerida (SIGIO en este caso) al proceso que cada manejador de archivo dijo que quería ser informado sobre el excitante nuevo movimiento del ratón que acaba de ocurrir.

Con esto en su sitio y arreglados los fallos en la versión original, tienes ahora un controlador de ratón totalmente funcional usando el protocolo del bus del ratón. El trabajará con X window system, trabajará con GPM y debería de trabajar con todas las otras aplicaciones que necesites. Doom es, por supuesto, la forma ideal para probar que tu nuevo controlador de ratón está funcionando de forma adecuada. Asegúrate de probarlo de todas las formas posibles.

Chapter 5. Sobre la traducción

Este documento es la traducción de "Mouse Drivers", documento que acompaña al código del núcleo de Linux, versión 2.4.18.

Este documento ha sido traducido por Rubén Melcón <melkon@terra.es>; y es publicado por el Proyecto Lucas (http://lucas.hispalinux.es)

Versión de la tradución 0.04 (Julio de 2002).

Si tienes comentarios sobre la traducción, ponte en contacto con Rubén Melcón <melkon@terra.es>